IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 6, 2019, accepted November 7, 2019, date of publication November 22, 2019, date of current version December 6, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2955282

Colored Petri Net Based Cache Side Channel Vulnerability Evaluation

LIMIN WANG, ZIYUAN ZHU, ZHANPENG WANG, AND DAN MENG
Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Corresponding author: Ziyuan Zhu (zhuziyuan@iie.ac.cn)

This work was supported in part by the Grand Science and Technology Program (Core Electronic Devices, High-end General Chips, and Infrastructure Software Products) under Grant 2018ZX01028201.

ABSTRACT The cache side channel leakage is a very serious security issue in the information security field. In order to solve this problem, a large number of security mechanisms have been applied to protect the cache. However, there are very limited methods we can choose to evaluate the cache side channel vulnerability, therefore, it is hard to know whether our system configuration or applied security mechanisms make caches more resistant to the cache side channel attacks. In this paper, we proposed a colored Petri net based method to model and score the cache side channel vulnerability. When given a side channel attack and related security mechanisms, our method utilized colored Petri net to model the requirements and the attack steps of cache attacks. Then we calculated the probability of success for each attack step according to the requirements and the computer environment, and the *Common Vulnerability Scoring System (CVSS)* was used to help us score the attack steps. Based on these probabilities and CVSS scores, we finally obtained a total risk score, which represented the threat level of the cache attacks in a specified computer environment with certain security mechanisms. This paper focused on the typical cache attacks and security mechanisms, and our experiments showed that we can conveniently evaluate and compare the threat level of cache attacks in the computer environment with different security mechanisms.

INDEX TERMS Cache side channel attack, security evaluation, CVSS, colored Petri net, qualification.

I. INTRODUCTION

Encryption is usually used to prevent confidential data leakage, and the ciphers generated by the complex encryption algorithm are difficult to crack through traditional cryptanalysis techniques [1]. Unlike cryptanalysis, side channel attacks are able to break the cipher by collecting and analyzing the signal released during data encryption [2]–[4]. In modern computers, in order to shorten the program's memory access latency, CPU caches will store a copy of recently used memory data so that the program can get the data more quickly. A *cache hit* means the requested memory data can be found in the cache, otherwise the requested data have to be read from memory, which is called *cache miss*. Due to the memory's low speed, the *cache miss* usually need much more extra clock-cycles than *cache hit*. Based on the difference of access time between *cache hit* and *cache miss*, cache based side channel attacks can recover the secret key by monitoring the cache access during the execution of an encryption program. Side channel attacks on caches are difficult to defend because the *cache hit* and *cache miss* they exploit are hardware features of CPU caches. In addition, it is important to mention that the past year has seen the rapid development of hardware vulnerabilities such as Meltdown [5] and Spectre [6], which extend traditional cache side channel attacks and make them more powerful. Combined with hardware vulnerabilities, the new cache side channel attacks are able to leak the victim's data directly. Fortunately, cache side channel attacks usually have some limitations, they rely on a variety of prerequisites such as *clflush* instruction or shared memory [7], that means computer or cloud system with different system configurations or security mechanisms have different ability to resist side channel attacks. Then there are two questions: (1) Are there certain system configurations or security mechanisms can make the attacker difficult to perform successful attacks? (2) Can we know the threat level of different cache side channel attacks on a computer system with a specified configuration or security mechanism?

The associate editor coordinating the review of this manuscript and approving it for publication was Xiangxue Li.

Different from software, as a computer hardware component, the CPU cache cannot be patched easily. Besides, caches play an important role in improving computer performance, removing caches from a computer will significantly degrade the performance. Therefore, if we can answer the question (1), we will be able to choose the most effective security mechanisms so that we can reduce the risk of side channel leakage with low cost. For example, rather than disabling all of the CPU caches, cloud providers can greatly improve the ability against cache side channel attacks by just modifying some system configurations such as disabling the simultaneous multithreading technique (SMT) on CPU or shared memory between cloud consumers [8]. And if we can answer the question (2), we can know which cache side channel attack is the most threatening to the current system with the specified configurations or security mechanisms so that we are able to prioritize high-risk cache side channel attacks.

There are some works try to answer these questions. Some approaches such as Side-Channel Vulnerability Factor (SVF) [9] and Cache Side-channel Vulnerability (CSV) [3] have been done to quantify the cache side vulnerability by running victim programs on a simulator and measure the difficulty of data leakage based on the collected data from the simulator. The main idea is to collect the victim's information during the victim's program execution and the attacker's observed side channel information in simulators, then we can calculate their Pearson correlation coefficient. The higher the coefficient, the higher possibility that the attacker can successfully recover the victim's secret key by analyzing the side channel information. The simulators based cache side channel vulnerability quantification methods mentioned above are easy to operate, however, it takes too much time to collect and write attack programs, in addition, performing the experiment and collecting necessary data are also slow [10]. To solve this problem, other works use the abstract cache model to analyze the side channel vulnerability, abstract models contain and only contain the essential features of attacker, victim, and security countermeasures. So on the one hand, they can also be used to represent the different attacks and defenses, and on the other hand, compared with the simulation, collecting necessary information through model analysis is more quickly. Zhang and Lee [11] propose a cache model for secure cache architecture, and use model checker *Murphi* to verify whether all of the information flow in the cache model obeys the Bell-LaPadula (BLP) policy by exhausting all of the explicit states. Further, deep learning is introduced by Zhang *et al*. [10] to analyze the relationship between the attacker's observed traces and victim's execution traces, these traces are collected from cache attack model. Then based on the training result, we can try to predict the victim's secret information from the side channel traces observed by attackers. He and Lee [12] use the probabilistic information flow graph to systematically model and evaluate the cache with different security mechanisms under different cache side channel attacks. Whereas their models have many advantages, however, they do not consider the system environment cache side channel attacks rely on and the threat level difference between different attacks.

In this paper, we focus on typical cache side channel attacks such as *Flush* + *Reload*, *Evict* + *Time*, and *Prime* + *Probe*. The combinations of these traditional attacks and hardware vulnerabilities such as Meltdown and Spectre will also be considered in our method. We propose a new quantitative approach to score the degree of threat of cache attacks in the computer environment with different security mechanisms. Our method mainly contains three phases: (1) We conclude the attack patterns of cache side channel attacks and their dependencies firstly. Then the model of cache side channel attacks will be built with colored Petri net. (2) We analyze the attack steps in different cache side channel attacks, then quantify the probability of success for each attack step based on their dependence and operations. The security weight of every attack step is also calculated according to the *Common Vulnerability Scoring System (CVSS)*. (3) Based on the probability and weight, we finally calculate the total risk score of the cache attacks. With this quantitative method, we can answer the first question that what security mechanisms are more resistant to the cache side channel attacks and the second question that what is the threat level of different cache side channel attacks in the computer environment with specified system configurations or security mechanisms.

In summary, the contributions this paper makes as follows:

1. We propose a new model based quantitative method to evaluate the threat of different cache side channel attacks in the computer environment with different security mechanisms. To make our evaluation approach more reasonable, in addition to the attack method, we also considered both the conditions on which the attack steps depend and the difference in attack capability between different attacks.
2. To qualify the threat level of different cache attacks, we propose a new three-step cache attacks model. In this paper, we divided the cache attacks into three steps, then CVSS was adopted to score the attack power of each attack step as the weight, we also analyzed the attack methods and their requirements to obtain the probability of success of every attack step. The attack steps and both the probability and weight will finally be modeled as a three-step colored Petri net model.

II. BACKGROUND

A. CACHE SIDE CHANNEL ATTACKS

Cache side channel attacks usually leak important data in the cache by measuring and analyzing the time of the victim's memory access. Access based cache side channel attacks record the time of victim's every memory access, then the attacker can distinguish the *cache hit* or *cache miss* from other memory access, these *cache hit* or *cache miss* we are interested in will reveal critical information such as access pattern and where the data is stored in the cache. In contrast, in timing based cache side channel attacks, the attacker usually performs some operations on the cache, and then

respectively record victim's total memory access time before and after the operations, the time difference before and after can also give away a lot of important information.

Most of the cache side channel attacks can be divided into three attack steps [13], in this paper, we will focus on four typical traditional cache side channel attacks and two new cache attacks. New cache attacks will be described in the section II-B, and the traditional attacks will be introduced in detail as below.

1) FLUSH+RELOAD

Flush+Reload is a high resolution access based cache side channel attack, this attack needs shared memory between the attacker and the victim, and it is able to attack last level cache [14], [15]. *Flush+Reload* can be divided into three attack steps:

1. The attacker carefully chooses the memory addresses and flushes these addresses with *clflush* instruction, then the cache lines mapped from the flushed addresses will be invalid.
2. Then the attacker waits for the victim to read or write memory at the flushed address.
3. The attacker tries to access the memory addresses chosen in step 1 and record the time of every access, if the access speed is fast, that means the current accessed memory address has been accessed by the victim in step 2, and the accessed data is cached.

2) EVICT+RELOAD

Clflush instruction exists in X86 so that the attacker can flush the cache easily. However, not all of the instruction set architectures (ISA) have the similar instruction, if the target of the attacker does not have *clflush* instruction, *Evict+Reload* can be used to replace *Flush+Reload* [16]. *Evict+Reload* is almost the same as *Flush+Reload*, and it also needs shared memory, the difference is *Evict+Reload* exploits cache replacement policies rather than *clflush* instruction to evict target cache lines. *Evict+Reload* has following steps:

1. Every memory address will be mapped to a cache set, when the cache set has no empty cache lines, the old cache lines will be replaced by new mapped cache lines according to the cache replacement policies. In this step, the attacker will try to frequently access the memory addresses that mapped to the same cache set as the victim's so that the victim's cache lines are evicted from the cache.
2. The attacker has to wait until the victim reads or writes memory at the evicted address again.
3. Same as the third step of *Flush+Reload*, the attacker will re-access the chosen memory addresses and record the access time.

3) PRIME+PROBE

Prime+Probe is also an access based cache side channel attack [17], [18], the attack steps are shown as follows:

1. In the *Prime* stage, the attacker firstly writes the specified data to the carefully chosen memory addresses, and the data will populate some cache sets.
2. Then the attacker waits for the victim to perform the encryption operation, if the victim's data is stored in the cache lines where the attacker's primed data is previously stored and then the primed data will be evicted.
3. Finally, the attacker reads data from the chosen memory addresses again, if the access time is long, that means the primed cache lines mapped from the accessed memory address have been replaced by victim's cache lines in the second step.

4) EVICT+TIME

Evict+Time is a timing based cache side channel attack [12], [17]. Different from the access based cache attacks, timing based attacks will measure the total time of the victim's operations. *Evict+Time* also has 3 steps:

1. The attacker waits for the victim to perform all of the operations, and record the total execution time.
2. Then the attacker will evict some victim's cache lines.
3. Finally, the attacker waits for the victim to perform the operations again, if the victim uses the cache lines evicted in step 2, the total execution time will be increased, and the attacker will observe this phenomenon.

B. CACHE SIDE CHANNEL ATTACK WITH HARDWARE VULNERABILITIES

Some hardware vulnerabilities such as *Meltdown* and *Spectre* are able to extend the cache side channel attacks and increase the attacker's capabilities. As II-A shown, in the second step of cache attacks, the attacker usually needs to wait for the victim to perform some important operations. However, with *Meltdown* and *Spectre*, the attacker does not have to wait for the victim anymore.

1) CACHE ATTACKS WITH SPECTRE

Spectre is a hardware vulnerability that exploits branch predictor vulnerabilities [6]. *Spectre* attack will train the branch predictor, and carefully make a wrong branch prediction that can be exploited by the attacker, then the processor will speculatively execute the attack program. During the speculative execution stage, the attacker can access the victim's data illegally and make them cached. With *Spectre* attack, if the second step of cache attacks needs the victim to access the memory, they can use *Spectre* attack instead.

2) CACHE ATTACKS WITH MELTDOWN

Meltdown is a hardware vulnerability that exploits out-of-order execution. In *Meltdown* attack, the attacker try to load the kernel data to a register, if the processor executes the instructions in order, the *load* operation will be denied due to the access permission violations. However, the out-of-order execution feature will make the illegal *load* operation be executed ahead of other normal instructions in front of it.

When the result of the *load* operation is committed, it is found that the *load* operation has violated the permissions, then the CPU raises an exception, so the kernel data will not be loaded into the target register. But unfortunately, the kernel data has been cached during out-of-order execution, and the cached data is not cleared when the exception is raised. Therefore, same as the *Spectre*, the second attack step in some of the cache attacks can be replaced by *Meltdown* attack.

C. CACHE SIDE CHANNEL SECURITY MECHANISMS

To defend these cache side channel attacks, different security mechanisms are proposed according to the characteristics of the attacks. For example, more and more cloud providers enhanced isolation between different customers, which will reduce shared memory. And some operating systems do not offer high-resolution timer, which will make the cache timing exploitation extremely challenging. Besides, some secure cache architectures such as Static Partition cache (SP cache) are also proposed by researchers.

Shared memory is usually used to reduce replicated content in memory, when two different processes access the same shared memory such as shared libraries, there will be some security problems which can be exploited to leak data. *Flush + Reload* and *Evict + Reload* are the typical attacks that exploit the shared memory. Fortunately, there is not always exploitable shared memory between the attacker and the victim, and some security isolation mechanisms like Intel SGX have been proposed to improve security [7], [19], [20].

Clflush instruction is an X86 instruction that can invalidate the cache lines of specified memory addresses in the whole cache hierarchy. In cache attacks, the *clflush* instruction can be used to flush some cache lines, some ISA does not provide similar instruction, which makes them more resilient to the *Flush + Reload*.

Evict strategy exploits continuous memory access to evict the victim's cache lines, which are often used in *Evict+Reload*, *Evict+Time*, and *Prime+Probe*. To address this security issue, some researchers proposed some new secure cache architecture such as SP cache and Set Associative (SA) Cache with the random replacement policy [12]. SP cache will statically isolate the attacker and victim and do not allow the attacker to interfere with the victim in the cache. Random replacement policy will make a specified address is able to randomly map to any cache lines in the cache set, which will make the attacker more difficult to evict target cache lines. Besides, the operating system and hypervisor can use Intel Cache Allocation Technology (CAT) to lock cache ways so that these cache ways cannot be evicted [11], [21], however, the attacker will still have a *cache hit* if the mapped cache lines are in the locked ways.

All of the cache side channel attacks need the high resolution timer to measure the time of cache accesses, thus some operating systems or browsers provide a coarse timer only to enhance security. But Schwarz *et al*. propose that the attacker can utilize other timing primitives to build a new high resolution timer [22], and Vasilikos *et al*. show that coarse timer may leak more information by analyzing the information flow model of the attacks [23].

D. COLORED PETRI NET

Colored petri net is a powerful tool to help us model and analyze the complex system. We choose it as the modeling technique in our paper for several reasons:

1. There are a variety of cache side channel attacks, however, three attack steps are enough to express them [13]. Colored Petri Net supports *hierarchical* feature so that we can build a three-step model as the top-level module which can be reused by the cache attacks, and the detail attack steps of different attacks can be modeled in the different submodules.
2. We need to model the dependence relations that when all of the requirements of an attack operation are satisfied, the operation can be triggered, and only when the current attack step is successful, can the next attack step be executed. Colored Petri Net provides *token* feature, which can help model the dependence relations in cache attacks.
3. We need to qualify the cache side channel vulnerability by assigning weights to different attack steps. Colored Petri net allows weight allocation and offers simulation and calculation functions.

In a colored Petri net, the *Place* are depicted as ellipses, and the *Transition* are depicted as rectangles. There is an input directed edge from a *Place* to a *Transition* and then an output edge connects the *Transition* and another *Place*. The *Place* represent the states of the model, and *Place* usually contains several tokens, these tokens have three properties: the number of tokens, data value, and data type, note that in a *Place*, all of the tokens have the same data type. The *Transition* represent the behaviors of the model, if the tokens in the input *Places* of the current *Transition* are available, and the *guard function* of the *Transition* is *true*, the *Transition* will be enabled, note that the *guard function* of the *Transition* is default to *true*.

Fig. 1 and Fig. 2 show a hierarchy colored Petri net example, Fig. 1 is a top-level module, and *T2* in the Fig. 1 is a submodule, which is show in Fig. 2.

In Fig. 1, *P1* is a *Place* with a token, *(1,"OK")* is the data of the token, and data type is *INTxDATA*, and *P2* also has a token. *T1* will be enabled because the token of *P1* and *P2* are available, then *T1* will consume the token in *P1* and *P2*. Note that the output edge of the *T1* has an expression, the token of *P1* and *P2* will be handled by the expression and then generate a new token for *P3*. And in Fig. 2, *P3* is the input port and *P4* is the output port, they are the interface between top-level module and submodule.

E. COMMON VULNERABILITY SCORING SYSTEM

Forum for Incident Response and Security Teams (FIRST) proposes the *Common Vulnerability Scoring System(CVSS)*, which is a framework to score the severity of the vulnerability [24], [25]. CVSS contains three dimensions: *Base Score*, *Temporal Score*, and *Environment Score*. *Base Score* consists

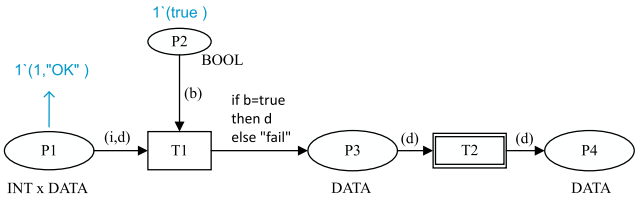


FIGURE 1. A sample: The top-level module of a hierarchical colored Petri net.

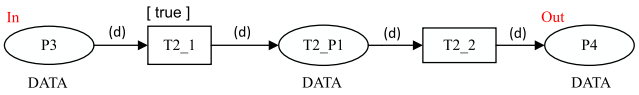


FIGURE 2. A sample: The submodule *T2* in Fig. 1.

of several metrics which are related to the inherent characteristics of the vulnerability, therefore *Base Score* is constant regardless of time and environmental changes. Based on the *Base Score*, *Temporal Score* and *Environment Score* will more accurately reflect the severity of vulnerabilities over time and the environment.

Due to the diversity metrics and accuracy of vulnerability severity evaluation, *CVSS* is widely used in information security. For example, National Vulnerability Database (NVD) uses *CVSS* to score the severity of vulnerabilities listed in Common Vulnerabilities and Exposures (CVE).

In this paper, we focus on the *Base Score*. As Table 1 shown, *Base Score* has two metrics and one scope, *Exploitability Metrics* has four characteristics, and they represent the ease of exploiting the vulnerabilities, *Impact Metrics* has three characteristics, they reflect the impact of a successful attack. *Scope* means whether the vulnerabilities can make attackers affect the resources that exceed attackers' authority.

Every attack step will be scored according to the CVSS specification [25]. We can analyze the *Attributes* of the attack step and the vulnerability the attack step exploits, then choose the *Value* in Table 1, and finally, we could get the security weight by using CVSS calculator [26].

Note: Why can we use CVSS represent the threat level of the attack steps?

It is an important question, we answer this question by answering another two smaller sub questions.

Sub question 1: The first sub question is that CVSS score represents the severity of the vulnerabilities, can it be used to represent the threat level of cache attacks?

Answer: CVSS score is able to represent the severity of vulnerabilities, which means the possible damage caused by the vulnerabilities when they are exploited [27], and the threat level of the attack also indicates the potential harm the attacks cause when they exploit the vulnerabilities. And the metrics of CVSS is also suitable for the attack steps, take *Exploitability Metrics* as an example, when a vulnerability is hard to exploit, the CVSS score of the vulnerability will be low, and for the same reason, if the exploitability of the vulnerability is low, the attack will also be difficult to perform, and the threat level of the attack will also be lower. Therefore, the CVSS score of a vulnerability can also reflect the threat level of attacks that exploit this vulnerability.

TABLE 1. Metrics and score in base score.

Metrics	Attributes	Value
Exploitability Metrics	Attack Vector (AV)	Network (N) Adjacent (A) Local (L) Physical (P)
	Attack Complexity (AC)	Low (L) High (H)
	Privileges Required (PR)	None (N) Low (L) High (H)
	User Interaction (UI)	None (N) Required (R)
Scope	Scope (S)	Unchanged (U) Changed (C)
Impact Metrics	Confidentiality Impact (C)	None (N) Low (L) High (H)
	Integrity Impact (I)	None (N) Low (L) High (H)
	Availability Impact (A)	None (N) Low (L) High (H)

In this paper, we will use CVSS metrics to help us score the attack steps of cache side channel attacks, and the detail will be shown in the section III-B.

Sub question 2: The second sub question is that some attack steps of cache attacks cannot leak any information or break the computer system, why the CVSS can also score them in this paper?

Answer: Some attack steps such as flushing the cache with *clflush* instruction will not cause the direct loss unless it is combined with other attack steps. The attack step seems harmless, but actually flushing the cache is very important for *Flush+Reload* attack, without this attack step, *Flush+Reload* will not succeed. In this paper, we think and analyze the attack steps more deeply, and we observed that there are important but not particularly serious security issues in the attack steps. Take *Flush+Reload* as an example again, in the first attack step, the attacker performs *clflush* instruction to flush victim's cache lines, which make *cache miss* occurs when the victim re-accesses the cache lines. The attack consumes the cache lines, therefore, the first step of *Flush+Reload* will actually have an impact on the availability of the victim's cache lines, and *Availability* is an attribute of CVSS metrics, thus we can also use CVSS to score the attack step. In the same way, we can score all of the attack steps, which will also be introduced in detail in the section III-B.

III. METHOD

The colored Petri net based cache side channel security evaluation method will be introduced in detail in this section. Fig. 3 shows a top-level module of hierarchical colored Petri net, which represents a cache side channel attack model. *Attack Step 1*, *Attack Step 2*, and *Attack Step 3* are the submodule which includes the attacker's attack operations. In the cache side channel attacks, if the *Req1*, *Req2*, *Req3*, and *Req4* of the attack step 1 are satisfied, the attacker will firstly perform the attack step 1. Then the attacker will try to perform the second attack step, similarly, only when the attack step 1 is successful and the *Req5* is satisfied, can the attack step 2 continues. After the attack step 2, the attacker will go to the next phase, and finally, if all of the *Req* of the step 3 are satisfied, the attacker will perform the final attack step. When the attack finishes, the attacker is able to observe the victim's confidential information leaked from the cache, which means this attack is successful.

Our method mainly includes the calculation of the probability and weight. The *Probability* part consists of the probability of successful attack operations performed by the attacker (PAO), the probability of attacker's successful attack steps (PAS), and the probability of success of the entire attack path (PAP). The *Weight* part contains the threat level of every attack step (WAS), if the attack step is successful, WAS will be the CVSS score of the attack step. Based on the probability and the weight on the cache side channel attack model, we will calculate and obtain a total score *RiskScore*, which represent the threat level of a cache attack in a certain computer environment.

A. THE PROBABILITY OF A SUCCESSFUL CACHE ATTACK

In the section II-A and II-B, we introduce that both the traditional cache attacks and the new cache side channel attacks with hardware vulnerabilities can be divided into three steps. Note that only when the first attack step is successful, can the second attack step be successful, and the successful second attack step is also necessary for the third attack step. Take *Flush + Reload* for example, the first step is flushing the cache, and the second step is waiting for the victim to access the memory, only when the first step that flushing the cache is successful, and the specified cache lines are flushed, can the second step that waiting for the victim's accessing the memory be useful. If the first attack step fails, the state of the cache is not what the attacker expected, then the subsequent attack steps will also be unsuccessful. Therefore, the cache attack succeeds if and only if all of the three attack steps are successful. We assume that *Attack Step i* represents the event that *i*th attack step is successful, and then PAP can be calculated by (1).

$$PAP = P(Attack\ Step\ 3 \wedge Attack\ Step\ 2 \wedge Attack\ Step\ 1) \quad (1)$$

As mentioned above, the first attack step is necessary for the second attack step, and the second attack step is necessary for the third attack step. Therefore, denote PAS of the first step as $P(Attack\ Step\ 1)$ (abbreviated as $P(AS1)$), and when the first attack is successful, the PAS of the second attack step can be denoted as $P(Attack\ Step\ 2 \mid Attack\ Step\ 1)$ (or $P(AS2)$ for short). And when the first and second attack steps are successful, the PAS of the third attack step can be denoted as $P(Attack\ Step\ 3 \mid Attack\ Step\ 1 \wedge Attack\ Step\ 2)$ (abbreviated as $P(AS3)$), and according to the conditional probability rules, PAP can be calculated in (2).

$$\begin{aligned} PAP &= P(Attack\ Step\ 3 \wedge Attack\ Step\ 2 \wedge Attack\ Step\ 1) \\ &= P(Attack\ Step\ 3 \mid Attack\ Step\ 1 \wedge Attack\ Step\ 2) \\ &\quad * P(Attack\ Step\ 1 \wedge Attack\ Step\ 2) \\ &= P(Attack\ Step\ 3 \mid Attack\ Step\ 1 \wedge Attack\ Step\ 2) \\ &\quad * P(Attack\ Step\ 2 \mid Attack\ Step\ 1) \\ &\quad * P(Attack\ Step\ 1) \\ &= P(AS3) * P(AS2) * P(AS1) \end{aligned} \quad (2)$$

The PAS of each attack steps are equal to the probability of success of the *Attack Step i* submodule, we will take *Attack Step 1* for example to explain how to calculate the PAS of an attack step. Fig. 4 shows a sample submodule of *Attack Step 1*. Note that the requirements in the Fig. 3 such as *Req1*, *Req2*, *Req3*, and *Req4* will be the input of *Attack Step 1* submodule, and if the attack operations needs some of these requirements, the related requirements *Req* will be connected to the attack

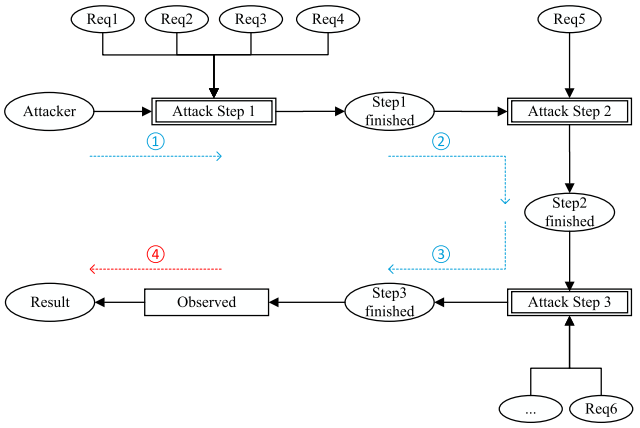


FIGURE 3. Hierarchical colored petri net of the cache side channel attacks.

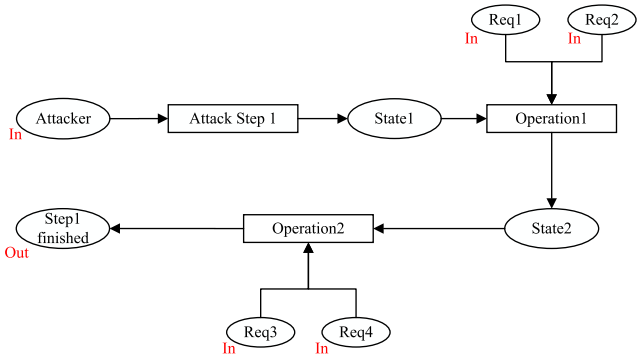


FIGURE 4. Attack module.

operation. For example, as Fig. 4 shown, the *Operation1* needs the requirements *Req1* and *Req2*, and then *Req1* and *Req2* will be connected to *Operation1*.

Under the premise that satisfying the requirements, every operation in the attack step has a probability of successful execution. For example, when the *Req1* and *Req2* are satisfied, the probability of *Operation1* can be easily calculated according to the computer environment and security mechanisms, the conditional probability can be denoted as $P(Operation1|Req1 \wedge Req2)$, and if one of the requirements of the *Operation1* is not satisfied, the *Operation1* will fail, and the probability $P(Operation1|Req1 \wedge Req2)$ will be 0.0. Then we can calculate the $P(Operation1)$ as following (3) according to the *Law of Total Probability*.

$$\begin{aligned}
&P(Operation1)\\
&= P(Operation1|\ Req1 \wedge\ Req2) \times P(\ Req1 \wedge\ Req2)\\
&\quad + P(Operation1|\neg(Req1 \wedge Req2)) \times P(\neg(Req1 \wedge Req2))\\
&= P(Operation1|\ Req1 \wedge\ Req2) \times P(\ Req1 \wedge\ Req2\\
&\quad + P(Operation1|\neg Req1 \wedge\ Req2) \times P(\neg Req1 \wedge\ Req2)\\
&\quad + P(Operation1|\ Req1 \wedge \neg Req2) \times P(\ Req1 \wedge \neg Req2)\\
&\quad + P(Operation1|\neg Req1 \wedge \neg Req2) \times P(\neg Req1 \wedge \neg Req2)\\
&= P(Operation1|\ Req1 \wedge\ Req2) \times P(Req1) \times P(Req2)\\
&\quad + 0 + 0 + 0\\
&= P(Operation1|\ Req1 \wedge\ Req2) \times P(Req1) \times P(Req2)
\end{aligned} \tag{3}$$

There is another situation, if *Operation2* has to satisfy the *Req3* or *Req4*, the PAO of *Operation2* will be as following (4), note that when both of the *Req* are not satisfied, the *Operation2* will also fail, and then the probability will decrease to be 0.0.

$$\begin{aligned}
&P(Operation2)\\
&= P(Operation2|\ Req3 \vee\ Req4) \times P(\ Req3 \vee\ Req4)\\
&\quad + P(Operation2|\neg(Req3 \vee Req4)) \times P(\neg(Req3 \vee Req4))\\
&= P(Operation2|\ Req3 \wedge\ Req4) \times P(\ Req3 \wedge\ Req4)\\
&\quad + P(Operation2|\neg Req3 \wedge\ Req4) \times P(\neg Req3 \wedge\ Req4)\\
&\quad + P(Operation2|\ Req3 \wedge \neg Req4) \times P(\ Req3 \wedge \neg Req4)\\
&\quad + P(Operation2|\neg Req3 \wedge \neg Req4) \times P(\neg Req3 \wedge \neg Req4)\\
&= P(Operation2|\ Req3 \wedge\ Req4) \times P(\ Req3 \wedge\ Req4)\\
&\quad + P(Operation2|\neg Req3 \wedge\ Req4) \times P(\neg Req3 \wedge\ Req4)\\
&\quad + P(Operation2|\ Req3 \wedge \neg Req4) \times P(\ Req3 \wedge \neg Req4)\\
&\quad + 0\\
&= P(Operation2|\ Req3 \wedge\ Req4) \times P(Req3) \times P(Req4)\\
&\quad + P(Operation2|\neg Req3 \wedge\ Req4) \times P(\neg Req3) \times P(Req4)\\
&\quad + P(Operation2|\ Req3 \wedge \neg Req4) \times P(Req3) \times P(\neg Req4)
\end{aligned} \tag{4}$$

After calculating the probability of the *Operation*, we can calculate the probability of the whole submodule *Attack Step 1*. The attack step is a sequence of the *Operation*, and these *Operations* are independent events, for example, in *Reload* step of *Flush+Reload*, there are two operations *re-access* and *measure time*, whether *re-access* operation occurs or not does not affect the probability of the successful occurrence of the *measure time* operation. Observed that to make the attack step successful, all of the *Operations* in the sequence should be successful, and these *Operations* have to follow a certain order. Assume the clever attacker will not mistake the order of the *Operations*, so the probability of the attack step as shown in (5):

$$P(ASi) = \prod_{i=1}^{n} P(Operation_i) \tag{5}$$

B. WEIGHT OF A SUCCESSFUL ATTACK STEP

The attack steps play an important role in contributing to a successful cache attack. In this paper, we utilize CVSS to evaluate the threat level of the attack steps, and the CVSS score of each step will be denoted as the *W(ASi)* .

In this part, we will introduce how to calculate the weight of every attack step based on the CVSS scoring system. According to the *Metric* in Table 1, we need analyze the *Attributes* of each attack step and choose a *value* that fit the attack step, the final result will be shown in the Table 2.

As shown in Table 2, all except *Spectre* step need run on the local machine, that is because the attacker have to share the cache with the victim so that they can successfully perform an attack to leak victim's data from the shared cache, thus the attribute *Attack Vector=Local* (AV:L). *Spectre* is special, which can be performed remotely [28], thus *Attack Vector=Network* (AV:N).

Spectre attack and *Meltdown* attack are a little complex, *Spectre* attack needs to train the branch predictors before exploiting the speculative execution, and *Meltdown* also needs to carefully design the code to create an exploitable out-of-order execution. Therefore, the *Attack Complexity* of both of *Spectre* and *Meltdown* are *High* (AC:H), and the *Attack Complexity* of other attack steps are *Low* (AC:L).

An attacker with user-level permissions can perform the cache side channel attacks, that is because cache side channel attacks do not utilize privilege level instructions. Therefore, all of the attack steps in the table require only user-level permissions (PR:L).

In *Flush+Reload*, *Evict+Reload*, and *Prime+Probe*, the second step has to wait for the victim to access the memory, therefore the *Wait* step needs user interaction (UI:R). In the first step and the third step of *Evict+Time*, the attacker needs to measure the total time of victim's operations, thus the *User Interaction* of both the first and third step are *Required* (UI:R). *Spectre* and *Meltdown* improve the attack power of traditional cache attacks, and they do not require the victim to take action in the second step of *Evict+Reload with Spectre*

TABLE 2. The CVSS score of attack steps.

Attacks	Attack Step	CVSS (Formula)	CVSS (Base Score)
Flush + Reload	Flush	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:H	5.5 (Medium)
	Wait	CVSS:3.0/AV:L/AC:L/PR:L/UI:R/S:U/C:L/I:N/A:N	2.8 (Low)
	Reload	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:L/I:N/A:N	3.3 (Low)
Evict + Reload	Evict	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:L	3.3 (Low)
	Wait	CVSS:3.0/AV:L/AC:L/PR:L/UI:R/S:U/C:L/I:N/A:N	2.8 (Low)
	Reload	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:L/I:N/A:N	3.3 (Low)
Prime + Probe	Prime	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:L	3.3 (Low)
	Wait	CVSS:3.0/AV:L/AC:L/PR:L/UI:R/S:U/C:L/I:N/A:N	2.8 (Low)
	Probe	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:L/I:N/A:N	3.3 (Low)
Evict + Time	Time	CVSS:3.0/AV:L/AC:L/PR:L/UI:R/S:U/C:L/I:N/A:N	2.8 (Low)
	Evict	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:L	3.3 (Low)
	Time	CVSS:3.0/AV:L/AC:L/PR:L/UI:R/S:U/C:L/I:N/A:N	2.8 (Low)
Evict + Reload with Spectre	Evict	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:L	3.3 (Low)
	Spectre	CVSS:3.0/AV:N/AC:H/PR:L/UI:N/S:C/C:H/I:N/A:N	6.3 (Medium)
	Reload	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N	5.5 (Medium)
Evict + Reload with Meltdown	Evict	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:L	3.3 (Low)
	Meltdown	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/S:C/C:H/I:N/A:N	5.6 (Medium)
	Reload	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/S:C/C:H/I:N/A:N	5.5 (Medium)

and *Evict+Reload with Meltdown*, so the *User Interaction* of *Spectre* step and *Meltdown* step is *None* (UI:N).

Not only *User Interaction*, *Spectre* and *Meltdown* are also able to improve the attack power of traditional cache attacks by accessing the victim's data illegally during the speculative execution and out-of-order execution, which means *Spectre* and *Meltdown* can affect the data that exceeds their access rights. Therefore, their *Scope* should be *Changed* (S:C), and the *Scope* of other attack steps is *Unchanged* (S:U).

In *Flush+Reload*, the *Flush* step uses *clflush* instruction to flush target cache line, which will make the victim's cache lines be unavailable to victim, therefore, the *Available* of the *Flush* step is *High* (A:H). *Evict* step needs the attacker to continuous access the memory so that they can evict the target cache lines, this method is not as effective as *clflush*, therefore the *Available* of the *Evict* step in the table is *Low* (A:L). Note that *Prime* also uses *Evict* strategy, and the *Available* of the *Prime* step is *Low* too. In *Flush+Reload*, *Evict+Reload*, and *Prime+Probe*, when the attacker *Wait* and the victim access the important data so that the data is cached again, there actually is some loss of confidentiality (C:L). In *Evict+Time*, when the attacker measure the total time in the first step and the third step, the *Confidentiality Impact* is also *Low*. However, in *Spectre* and *Meltdown*, they can illegally access and cache the data, therefore, their *Confidentiality Impact* is *High* (C:H). In the third attack step of *Flush+Reload*, *Evict+Reload*, *Prime+Probe*, and *Evict+Time*, the attacker can not obtain the victim's secret key directly, and the attacker has to calculate the secret key based on the observed side channel information. Therefore, the third step of these attacks can only make the victim suffers minor loss of confidentiality (C:L). Different from these traditional cache side channel attacks, in the third step of *Evict+Reload with Spectre* and *Evict+Reload with Meltdown*, the attacker can obtain the secret data cached in the second step, therefore, the *Confidentiality Impact* is *High* (C:H).

C. QUALIFICATION

According to the Part III-A and Part III-B, we have calculated the weight and the probability of each successful attack step. Based on these calculated results, we can get the score of the cache side channel risk, which is denoted as *RiskScore*, the *RiskScore* reflects the threat level of a cache side channel attack in the computer environment with specified security mechanisms.

Some attack steps are powerful, and their WAS are also high, but they may need more requirements. However, the probability of a given computer environment satisfying all of these requirements is low, which makes it have a low PAS value. In order to make the risk score of every attack step reflect PAS and WAS at the same time, we denote the risk score of *Attack Step i* as F_i, which can be calculated

in (6). $PAS = P(ASi)$ means the probability of success of *Attack Step i*, $WAS = W(ASi)$ means the damage caused by the successful attack step i, which is a CVSS score, and when the attack step fails, $WAS = 0$.

$$\begin{aligned} F_i &= P(ASi) \times W(ASi) + (1 - P(ASi)) \times 0 \\ &= P(ASi) \times W(ASi) + 0 \\ &= P(ASi) \times W(ASi) \end{aligned} \tag{6}$$

Then, according to (6), the risk score of the three attack steps can be calculated respectively according to following formulas (shown in (7), (8), and (9)).

$$F_1 = P(Attack\ Step\ 1) \times W(AS1) \tag{7}$$

$$F_2 = P(Attack\ Step\ 2 \,|\, Attack\ Step\ 1) \times W(AS2) \tag{8}$$

$$\begin{aligned} F_3 = {} & P(Attack\ Step\ 3 \,|\, Attack\ Step\ 1 \wedge Attack\ Step\ 2) \\ & \times W(AS3) \end{aligned} \tag{9}$$

Finally, the total *RiskScore* will be the sum of the attack steps' *RiskScore*, which can be shown in (10).

$$RiskScore = \sum_{i=1}^{3} F_i \tag{10}$$

IV. CASE STUDY AND EXPERIMENTS

A. CASE STUDY: EVICT+RELOAD WITH SPECTRE

Evict+Reload with Spectre attack is a new cache side channel attack that contains the characteristics of traditional side channel attack and hardware vulnerability. Therefore, we take *Evict+Reload with Spectre* as an example in this section to explain how to calculate the total *RiskScore* in detail, and then we also give an analysis of other five cache side channel attacks. We assume the default computer environment in this paper is an Intel processor with 4-way set-associative cache.

Algorithm 1 shows the *Evict+Reload with Spectre* attack which is modified from the source code of the *Spectre* attack [6]. From line 9 to line 12, the attacker usually allocates a huge array (larger than cache size), and then continuously writes data to the array from the beginning of the array to the end of the array, then previous cache lines will be evicted from the cache.

In line 16, *tries* means the attacker tries the Algorithm 1 for the (*tries*)th time. The codes from line 17 to line 19 means that the attacker trains the branch predictor by normally accessing the *array1* five times (when $i\%6 \neq 0$), and then executes the attack operations once (when $i\%6 = 0$). During the sixth loop, line 18 will be executed, in line 18, *secret_addr* is the address of victim's data. As we all known, $array1[i] = *(array1+i)$, and which means the *(i+1)*th element of *array1*. Therefore, *secret_address* can be described as an address that its base address is *array1* and offset address is $tmp\_index = secret\_addr - arry1$. And then, *the data at the secret_addr* $= *(secret\_addr) = *(array1 + secret\_addr - array1) = array1[secret\_addr - array1] = array1[tmp\_index]$.

Algorithm 1 Evict+Reload With Spectre

Input:

STRING * secret_addr;
/*the address of target data.*/

Output:

CHAR secret_data;
/*the leaked target data .*/

```
1:  /* Init Attack */
2:  INT64 time1=0, time2=0;
3:  /*64 bit integer variables */
4:  INT8 array1[16]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};

5:  /*8 bit integer variable */
6:  INT8 array2[256];
7:  /*8 bit integer variable */
8:
9:  /* Attack Step 1: Evict */
10: Allocate a large size array;
11: Fill the array with new data;
12: Victim's cache lines will be evicted from the cache;
13:
14: /* Attack Step 2: Spectre Attack */
15: for i = 29 to 0 do
16:     tmp_index= tries % array1_size;
17:     if i % 6==0 then
18:         tmp_index = secret_addr − array1 ;
19:     end if
20:
21:     if  tmp_index < array1_size then
22:         tmp_data &= array2[array1[tmp_index]]
23:     end if
24: end for
25:
26: /* Attack Step 3: Reload */
27: for i= 0 to 255 do
28:     time1 = RDTSCP;
29:     Read array2[i];
30:     time2 = RDSTCP;
31:     if time2 − time1 < CACHE_HIT_THRESHOLD
        AND i != array1[tries % array1_size] then
32:         return CHAR(i)
33:     end if
34: end for
```

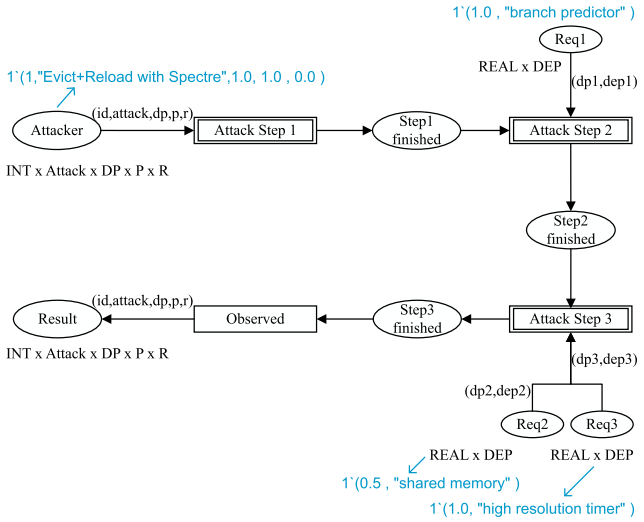


FIGURE 5. The attack steps of *Evict+Reload with Spectre* attack.

From line 21 to line 23, the results of the branch predictor in the first five loops are *Taken* and the real branch results are also *Taken*. After training the branch predictor, at the sixth loop, the result of the branch predictor will still be *Taken*, which means line 22 will be executed in the speculative execution phase. In line 22, *array1[tmp_index]* is the data at the *secret_addr*, then the data will be used as the index of the array2, when access the *array2[*secret_addr]*, the data at the *secret_addr* will be cached. However, actually, in the sixth loop, the *tmp_index* has been changed to the victim's offset address *secret_addr* − *arry*1 in line 18, thus the branch operation in line 21 will not actually be *Taken*, and line 22 will also not be executed. When the CPU finds the result of the branch predictor is not correct, the result of speculative execution will not be committed. From the perspective of the programmer, line 22 in the sixth loop is never executed, but from the perspective of the micro-architecture, line 22 has been executed, but finally, the result of execution is discarded. However, the data at *secret_addr* has been cached during the speculative execution is still in the cache.

From line 28 to 30, the attacker will record the time before and after re-accessing. When accessing *array2[i]*, if the access time *time2* - *time1* is less than *CACHE_HIT_THRESHOLD*, which means *cache hit* occurs, and *array2[i]* has been accessed in the attack step 2. Therefore, the index *i* in line 29 is equal to the *array1[tmp_index]* in line 22. Note that in *Attack Step 2*, *array1[tmp_index]=array1[tries%array1_size]* in the first five loops is the elements in *array1*, and in the sixth loop, *array1[tmp_index]=array1[secret_addr-array1]* is the data at the *secret_addr*, and it is not in *array1*. Therefore, in line 31, if the index *i* of the *array2* is not in *array1*, it is the cached secret data. line 32 will convert the secret data represented by ASCII code to a character and finally return the character.

Now we have leaked the first character of the secret data, and in this way, we can leak the whole secret data finally.

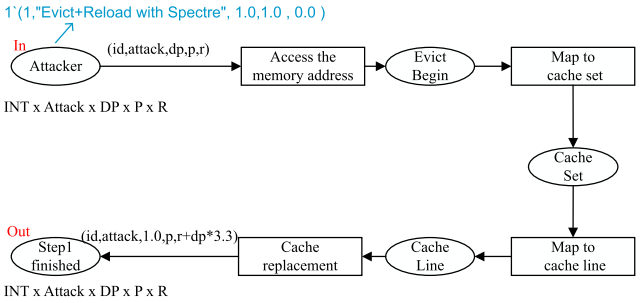


FIGURE 6. *Attack step 1* submodule of *Evict+Reload with spectre* attack.

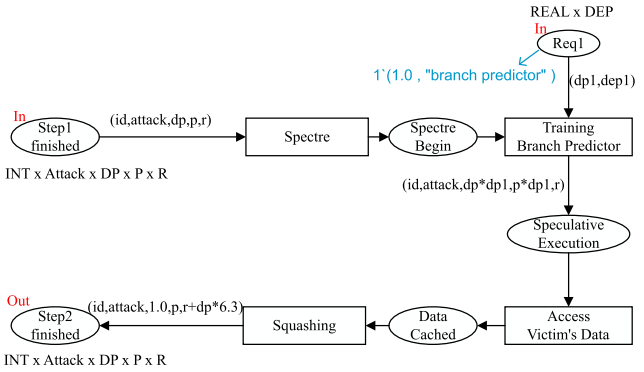


FIGURE 7. *Attack step 2* submodule of *Evict+Reload with spectre* attack.

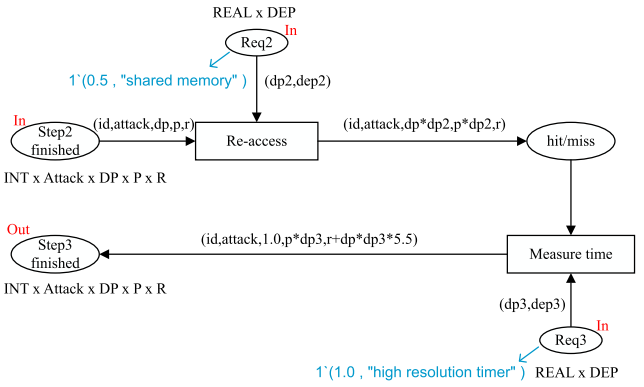


FIGURE 8. *Attack step 3* submodule of *Evict+Reload with spectre* attack.

However, cache side channel attacks usually need some requirements, one of the requirements is that the attack steps before the current attack step must be successful, and another is the system environment and security mechanisms. Fig. 5 is the top-level module for *Evict + Reload with Spectre* attack. Fig. 6, Fig. 7, and Fig. 8 are the attack step submodules of *Evict+Reload with Spectre*. In Fig. 5, *Attacker* has a token *(1,"Evict+Reload with Spectre",1.0, 1.0, 0.0)*, and the type of the token is *INT x Attack x DP x P x R. 1* in the token is of type *INT*, which means the sequence number of the token, and *Evict+Reload with Spectre* represent one of the *Attack*, *DP* means the PAS of every attack step, *P* and *R* are respectively representing the PAP and RiskScore of this attack. This token will be assigned to the varieties *id*, *attack*, *dp*, *p*, and *r* as

initial values, and then the transition *Attack Step 1* performs, the variables can be processed during the edge from the *Attack Step 1* to *Step 1 finished*, and the new value of the variables will be as a new token of *Step1 finished*. Notice that *Req1* also has a token *(1.0,"branch predictor")*, the value of the token will be assigned to the variable *dp1* and *dep1*, *dp1* represents the probability that *Req1* is satisfied (denoted as P(Req1)). Because the tokens in *Step1 finished* and *Req1* are available, *Attack Step2* also performs. Then the subsequent transitions perform in the similar way until the tokens reach *Result*.

As shown in Fig. 6, there are four operations in *Evict* step, the probability of the first two operations are 1.0. That is because for a specified virtual address, it will be mapped to a specific cache set. And because the accessed memory addresses are carefully selected, thus the selected address and target victim's address will be mapped to the same cache set so that the attacker can evict the victim's cache lines. In the widely used set-associative cache, a cache set has multiple cache lines, therefore we are not sure which cache line will the address map to in the third operation *Map to cache line*. If the mapped cache line is exactly a victim's cache line, and then the fourth operation *Cache replacement* will apply the cache replacement policy to replace the victim's cache line with the new mapped cache line. And if the mapped cache line is not the victim's cache line, then the attacker needs to try the third and fourth operations repeatedly until the victim's cache line is evicted successfully. Due to the LRU policy, when the number of trials is larger than the number of lines in a cache set, the victim's cache line will be bound to be evicted, therefore the probability of the latter two operations is also 1.0. So in the final edge which is between *Cache replacement* and *Step1 finished*, because this edge is the last one in an attack step, *dp* will be reset to 1.0, so that we can conveniently calculate the PAS of the next attack step. The probability p is not modified, and the risk score *F1* has been calculated and added to the result r.

In Fig. 7, there are four important operations in *Spectre* step. The second important operation *Training Branch Predictor* and third important operation *Access Victim's Data* are the code from line 15 to line 24 in Algorithm 1, and they can execute successfully only when the CPU supports the branch predictor. Note that the Intel processor used in this paper support the branch predictor, therefore we define the probability that the requirement *Req1* is satisfied as 1.0, when the requirement is satisfied, *Training Branch Predictor* can be achieved through the code from line 16 to line 19 in the Algorithm 1, therefore $P(Training\ Branch\ Predictor \mid Req1) = 1.0$, and the PAO of second important operation can be calculated according to (3), which means $P(Training\ Branch\ Predictor) = 1.0$. During the speculative execution, the attack operation *Access Victim's Data* can access the data illegally, and the PAO of this operation is also 1.0. The fourth important operation *Squashing* does not need any requirements, when the outcome of the branch predictor is not correct, the CPU is able to squash the speculative execution instructions successfully, so $P(Squashing) = 1.0$. Then $F_2 = dp * 6.3$ will be added to the result r in the last edge between *Step 2 Finished* and *Squashing*.

TABLE 3. The PAP and *RiskScore* of the *Evict+Reload with spectre* attack.

Security Mechanisms	P(AS1) / F1	P(AS2) / F2	P(AS3) / F3	PAP / RiskScore
No Security Mechanisms	1.0 / 3.3	1.0 / 6.3	0.5 / 2.75	0.5 / 12.35
No *clflush*	1.0 / 3.3	1.0 / 6.3	0.5 / 2.75	0.5 / 12.35
No shared memory	1.0 / 3.3	1.0 / 6.3	0.0 / 0.0	0.0 / 9.6
SP cache	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0
CAT	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0
SA cache (Random)	0.68 / 2.244	1.0 / 6.3	0.5 / 2.75	0.34 / 11.294
No high resolution timer	1.0 / 3.3	1.0 / 6.3	0.25 / 1.375	0.25 / 10.975

In the same way, as Fig. 8 shown, the operation *re-access* needs the *shared memory*, and there may not be available shared memory between the attacker and the victim, therefore we define the probability of *Req2* as 0.5, then $P(Re\text{-}access) = P(Re\text{-}access|Req2) * P(Req2) = 1 * 0.5 = 0.5$. Because the modern Intel processor and operating system provides *high resolution timer*, therefore the PAO of the operation *Measure time* can be calculated in formula $P(Measure\ time) = P(Measure\ time|Req3) * P(Req3) = 1.0 * 1.0 = 1.0$. Finally, $F3 = dp * dp3 * 5.5$ will be added to r in the last edge, then we obtain the *RiskScore* r and PAP p, the entire attack process can be simulated and calculated in the CPN tools [29], which is a powerful colored Petri net tool. The result is shown in the first row of Table 3.

Table 3 shows the PAP and *RiskScore* of *Evict+Reload with Spectre* attack before and after applying the security mechanisms. Because *Evict + Reload with Spectre* does not need the *clflush*, thus when the ISA does not have *clflush* instruction, the PAP and *RiskScore* are still the same with the attack in the system without any security mechanisms. Shared memory is necessary for *Evict + Reload* related attacks, when the cloud provider equips the system with security isolation mechanisms, the attacker will be hard to find the available shared memory, therefore, in the third step of the attack, the probability that *Req2* in Fig. 8 is satisfied will be 0.0, and then $P(AS3) = 0$, $F3 = 0$. The whole attack is not

successful due to the failure of the third attack step, $PAP = P(AS1)*P(AS2)*P(AS3) = 0$. Although the attack fails, it is still a dangerous attack, that is because, the attacker usually exploits different vulnerabilities to achieve the same attack step, the success of the first two steps may be exploited by other attacks, so it is reasonable that the unsuccessful attack still has a non-zero *RiskScore*.

With SP cache, the reason why the victim's cache lines are not allowed to be evicted is that the attacker's memory address will not be mapped to the victim's partitions (cache sets) so that the victim's cache lines will not be replaced by attacker's cache lines. That means in Fig. 6, the PAO of the operation *Map to cache set* will reduce to 0.0. Because the first attack step fails, and the requirements of subsequent attack steps will not be satisfied, therefore the probability of all of the attack steps will be reduced to 0.

CAT based methods are able to set the specified victim's cache ways as read-only, and the attacker can still hit at the protected way, however, the victim's cache lines in the protected ways will not be evicted by the attacker. Different from SP cache, CAT based security mechanisms will make the probability of *Cache replacement* in attack step 1 (shown in Fig. 6) to be 0.0. Then the probability of the subsequent attack steps will also be 0.0.

In modern SA cache, least recently used (LRU) is the commonly used cache replacement policy. When memory addresses are mapped to the cache lines, if there are no empty cache lines in mapped cache set, the cache will choose the least recently used cache line in the cache set, and then replace this cache line with the new mapped cache line. With the LRU policy, the attacker is able to evict a target cache line by accessing the memory address up to four times. In random replacement policy, when all of the cache lines in the cache set is full, the cache will randomly choose one cache line in the cache set to replace, which may reduce memory access performance, but it makes the cache more resilient to the cache side channel attacks, that is because the probability of evicting a target cache line within four accesses in SA cache with the random replacement policy will decrease to $\sum_{i=1}^{4}(\frac{3}{4})^{i-1} \times \frac{1}{4} \approx 0.68$.

Measuring the cache access time is a necessary operation in all of the cache attacks because the attacker has to differ *cache hit* and *cache miss* by analyzing the recorded time. The attack usually uses the system API *rdstcp*, it is nearly the most convenient and the highest precision method. Because the attack can use other methods to measure time, thus offering the coarser timer only is not a perfect security countermeasure, but it can increase the difficulty of the attack. In the attack step 3, we assume that when there is no high resolution timer, the attacker has another 50% probability to successfully construct a new timer, and we modify the probability of the *Req3* from 1.0 to 0.5.

B. OTHER EXPERIMENTS

In this part, we will show how to calculate the *RiskScore* of other cache side channel attacks. For convenience, in this part, the top-level module and submodules of cache attacks will be shown in one figure.

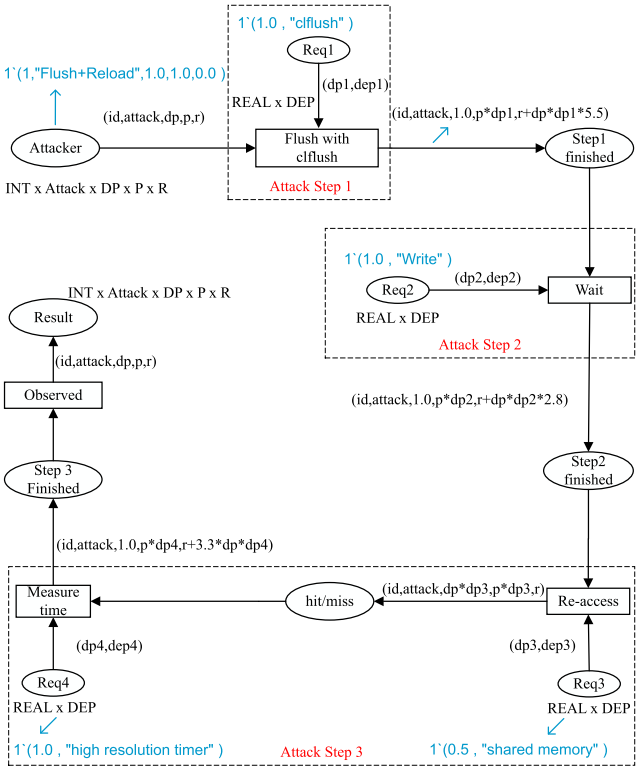


FIGURE 9. Colored petri net model of *Flush+Reload* attack.

1) FLUSH+RELOAD

Fig. 9 shows the Petri net model of *Flush+Reload* attack, and Table 4 shows the *RiskScore* of the attack under different security mechanisms. *Clflush* is an instruction to invalidate the specified cache lines, if the ISA has *clflush*, the flush operation can be successful, therefore $P(Flush\ with\ clflush|Req1) = 1.0$, and $P(Flush\ with\ clflush) = P(Flush\ with\ clflush|Req1) * P(Req1) = 1.0$. *Req2* is special, the attacker will keep performing the attack step 1 until the victim executes the expected access operation, thus $P(Req2) = 1.0$, and $P(Wait) = P(Wait|Req2) * P(Req2) = 1.0 * 1.0 = 1.0$. The *Re-access* operation in third attack step also needs shared memory, and *Measure time* needs high resolution timer, only when these two operations succeeds, can the third attack steps be successful. Similar to the third step of *Evict+Reload with Spectre*, $P(AS3) = P(Measure\ time) * P(Re\text{-}access) = P(Measure\ time|Req3) * P(Req3) * P(Re\text{-}access|Req2) * P(Req2) = 0.5$. Therefore, when there is no security mechanisms, the *RiskScore* of *Flush+Reload* is equal to 9.95.

Disabling the *clflush* instruction can help defend against *Flush+Reload* attack. Due to the lack of necessary *clflush* instruction, the first attack step will fail, then all of the subsequent attack steps will also fail, and their probability will be 0.0.

TABLE 4. The PAP and *RiskScore* of the *Flush+Reload* attack.

Security Mechanisms	P(AS1) / F1	P(AS2) / F2	P(AS3) / F3	PAP / RiskScore
No Security Mechanisms	1.0 / 5.5	1.0 / 2.8	0.5 / 1.65	0.5 / 9.95
No *clflush*	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0
No shared memory	1.0 / 5.5	1.0 / 2.8	0.0 / 0.0	0.0 / 8.3
SP cache	1.0 / 5.5	1.0 / 2.8	0.5 / 1.65	0.5 / 9.95
CAT	1.0 / 5.5	1.0 / 2.8	0.5 / 1.65	0.5 / 9.95
SA cache (Random)	1.0 / 5.5	1.0 / 2.8	0.5 / 1.65	0.5 / 9.95
No high resolution timer	1.0 / 5.5	1.0 / 2.8	0.25 / 0.825	0.25 / 9.125

When there is no shared memory between the attacker and the victim, the first and second attack step is the same as the attack steps when the computer environment has no any security mechanisms. But in the third step, *Re-access* operation will fail, then $P(AS3) = P(Measure\ time) * P(Re\text{-}access) = 0.0$ and PAP will also be 0.0.

The aim of SP cache and CAT based security mechanisms is to protect the victim's cache lines from been evicted, however, *Flush+Reload* do not need to evict the victim's cache lines, the attack use *clflush* instructions to make the victim's cache lines invalid. So both the SP cache and CAT technology cannot defend against *Flush+Reload* attack.

Similarly, SA Cache with the random replacement policy is able to effectively reduce the probability of successful evicting the victim's cache lines. However, *Flush+Reload* does not utilize *Evict* strategy. So SA Cache with the random replacement policy also cannot defend against *Flush+Reload*.

High resolution timer is an important tool for cache attacks. Without the precise timer, the system cannot completely resist the attack, but it can make the attack more challenging. When there is only a coarse timer, and the probability of building a new high resolution timer is 0.5, then $P(Measure\ time) = P(Measure\ time \mid high\ resolution\ timer) * P(high\ resolution\ timer) = 1.0 * 0.5 = 0.5$.

2) EVICT+RELOAD

In *Evict+Reload* (shown in Fig. 10), the first step is the same as *Evict+Reload with Spectre*, *Evict* does not need special requirements. Therefore, when given a computer system without security mechanisms, $P(AS1) = 1.0$. At the second step, the attacker is waiting for the victim to access the shared memory, and the third attack step is re-accessing the evicted memory, which are similar to the second step and the third attack step of *Flush+Reload*, then we can easily know that $P(AS2) = 1.0$ and $P(AS3) = 0.5$.

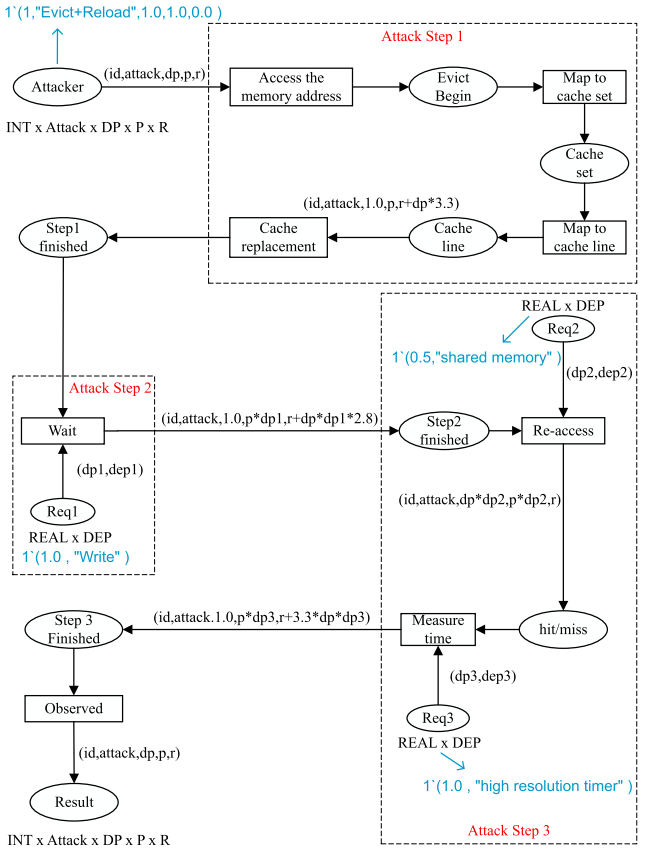


FIGURE 10. Colored petri net model of *Evict+Reload* attack.

Unlike *Flush+ Reload*, *Evict+Reload* attack do not use *clflush* instruction, therefore disabling the *clflush* instruction do not affect the *Evict + Reload* attack. However, *Evict+Reload* also needs shared memory, which is the same as *Flush+Reload* attack. Therefore, when there is no shared memory, $P(AS3) = 0.0$.

SP cache and CAT based protection technologies can prevent the external interference from the attacker [11], which can make the victim's cache unevictable. Therefore, in the first attack step of the *Evict+Reload* attack, the probability of success will be 0.0, then all of the PAS in this attack will become 0.0.

Similar to the *Evict+Reload with Spectre* attack, when the security mechanism is SA cache with the random replacement policy, the PAS of the first attack step will be 0.68 and when equipped with a coarse timer, the PAS of the third attack step will decrease to 0.25. The results mentioned above will be recorded in the Table 5.

TABLE 5. The PAP and *RiskScore* of the *Evict+Reload* attack.

Security Mechanisms	P(AS1) / F1	P(AS2) / F2	P(AS3) / F3	PAP / RiskScore
No Security Mechanisms	1.0 / 3.3	1.0 / 2.8	0.5 / 1.65	0.5 / 7.75
No *clflush*	1.0 / 3.3	1.0 / 2.8	0.5 / 1.65	0.5 / 7.75
No shared memory	1.0 / 3.3	1.0 / 2.8	0.0 / 0.0	0.0 / 6.1
SP cache	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0
CAT	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0
SA cache (Random)	0.68 / 2.244	1.0 / 2.8	0.5 / 1.65	0.34 / 6.694
No high resolution timer	1.0 / 3.3	1.0 / 2.8	0.25 / 0.825	0.25 / 6.925

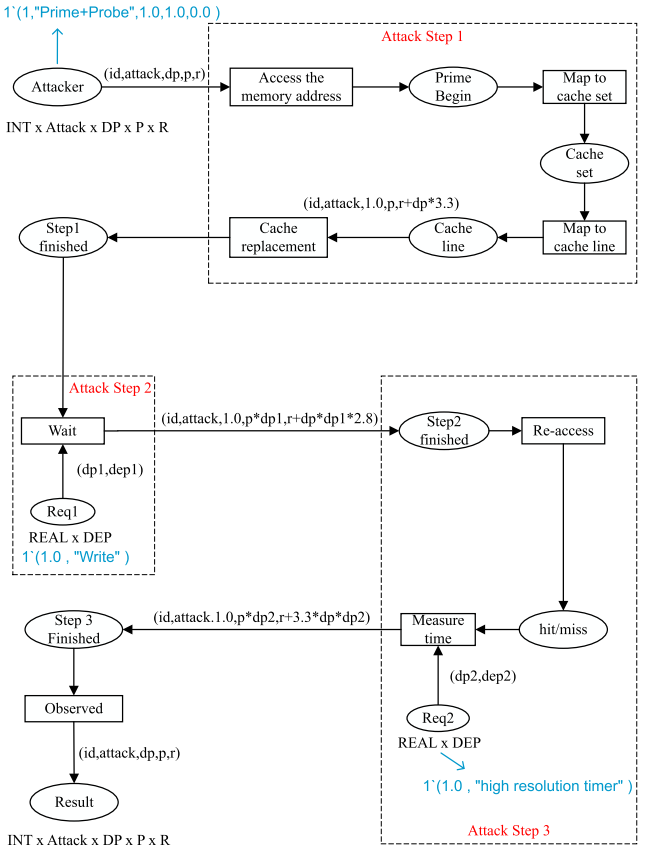


FIGURE 11. Colored petri net model of *Prime+Probe* attack.

3) PRIME+PROBE

As Fig. 11 shown, the colored Petri net model of *Prime+Probe* attack is nearly the same as *Evict+Reload*, which means they have similar attack operations in attack steps. However, there actually be some difference between *Evict+Reload* and *Prime+Probe*.

In the first attack step, both of them need to evict some cache lines, so the PAS of the first attack step of *Prime +Probe* is also 1.0. But actually the aim of the *Evict* step in *Evict+Reload* is to evict the victim's cache lines out of the cache, while the purpose of the *Prime* step is to make the attacker's cache lines occupied some cache lines so that the attacker can monitor the use of these cached lines.

In the second attack step, the attacker in *Evict+Reload* and *Prime+Reload* has to wait for the victim to access the specified memory address. Therefore, the PAS of the second attack step is also 1.0.

In the third attack step, these two attacks will re-access the selected address in the first attack step. However, *Evict+Reload* needs re-access the shared memory, but the *Prime+Probe* does not need. Therefore, the PAS of the third step in *Prime+Probe* is different from the *Evict+Reload*, and the PAS of the third step in *Prime+Probe* is 1.0. According to (10), $RiscScore = F_1 + F_2 + F_3 = 1.0 * 3.3 + 1.0 * 2.8 + 1.0 * 3.3 = 9.4$.

Prime+Probe do not need *clflush* instruction and shared memory, therefore the security mechanisms *No clflush* and *No shared memory* do not affect the attack power of *Prime+Probe* attack. And the *RiskScore* under the computer environment without *clflush* and *shared memory* is the same as the *RiskScore* under the computer environment without any security mechanisms.

SP cache and CAT based techniques can be used to effectively defend against external interference, as a result, the *Prime* step will not be successful. Then the PAS of subsequent attack steps will also be 0.0.

When using SA cache with the random replacement policy, the first attack step is special, the attacker has to evict all of the cache lines in a cache set. If the replacement policy of the SA cache is LRU, when the number of accessing the memory is larger than four, the specified cache set will be evicted successfully. However, if the SA cache equipped with the random replacement policy, to evict the cache set with four accesses, each access has to be mapped to different cache lines of a cache set, and the PAO of the operation *Cache replacement* will be 0.0039, which is calculated from the equation $PAO = 0.25 * 0.25 * 0.25 * 0.25 = 0.00390625 \approx 0.0039$. And then $F1 = 0.01287$, which is shown in $F1 = P(AS1) * W(AS1) = 0.0039 * 3.3 = 0.01287$.

With a coarser timer, PAO of the operation *Measure time* reduces to 0.5, and the PAS of the third attack step also reduces to 0.5. All of the results of PAP and *RiskScore* will be shown in Table 6.

TABLE 6. The PAP and *RiskScore* of the *Prime+Probe* attack.

Security Mechanisms	P(AS1) / F1	P(AS2) / F2	P(AS3) / F3	PAP / RiskScore
No Security Mechanisms	1.0 / 3.3	1.0 / 2.8	1.0 / 3.3	1.0 / 9.4
No *clflush*	1.0 / 3.3	1.0 / 2.8	1.0 / 3.3	1.0 / 9.4
No shared memory	1.0 / 3.3	1.0 / 2.8	1.0 / 3.3	1.0 / 9.4
SP cache	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0
CAT	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0
SA cache (Random)	0.0039 / 0.01287	1.0 / 2.8	1.0 / 3.3	0.0039 / 6.11287
No high resolution timer	1.0 / 3.3	1.0 / 2.8	0.5 / 1.65	0.5 / 7.75

4) EVICT+TIME

Fig. 12 are the colored Petri net model of *Evict+Time* attack, Table 7 is the *RiskScore* of the attack. In the first attack step, the operation *Measuring the total time* needs high resolution timer, the Intel processor and the operating system can provide the satisfactory timer. Therefore, $P(\textit{Measuring the total time}) = P(\textit{Measuring the total time}|Req1) * P(Req1) = 1.0 * 1.0 = 1.0$. And the PAS of the first attack step is 1.0, which is shown in $P(AS1) = P(Wait) * P(\textit{Measuring the total time}) = 1.0$.

In the second step, the attacker evicts the victim's cache lines, under the computer environment without security mechanisms, the PAS of this step is the same as the PAS of the first step of *Evict+Reload* attack. The PAS of the attack step is 1.0.

The third step is the same as the first step, the attacker needs to wait and measure the victim's total time again so that the attacker can observe that whether the victim uses the cache lines evicted in the second step. Therefore, $P(AS3) = P(AS1) = 1.0$. The *RiskScore* under the computer environment without any security mechanisms will be shown in Table 7.

clflush instruction and shared memory are not necessary for *Evict+Time* attack, therefore, the *RiskScore* of *Evict+Time* will not change if the computer environment does not have *clflush* instruction and shared memory. On the contrary, SP cache and CAT could effectively defend against *Evict* related attack steps, therefore if SP cache and CAT technology are applied to the computer environment, the second attack step will fail, and the PAS of the second attack step will be 0.0.

SA cache with the random replacement policy will randomly choose a cache line and replace them when the cache set is full, which makes the attacker difficult to evict target cache lines. With this security mechanism, the PAS of the second attack step decrease to 0.68. Besides, Using a coarser timer makes the attacker hard to measure time, and which will make the PAS of the first attack step and the third attack step reduce to 0.5.

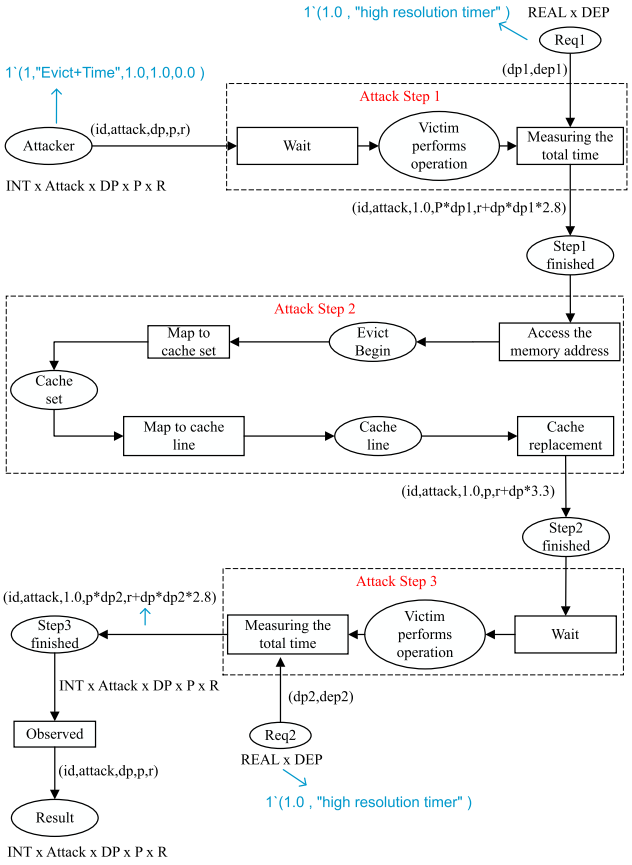


FIGURE 12. Colored petri net model of *Evict+Time* attack.

5) EVICT+RELOAD WITH MELTDOWN

Meltdown is another serious hardware vulnerability which exploits Out-of-Order feature of the modern processor. Fig. 13 shows the colored Petri net model of the *Evict+Reload with Meltdown* attack, and Table 8 shows the *RiskScore* of the *Evict+Reload with Meltdown* attack in the computer environment with different security mechanisms.

The first attack step *Evict* will carefully choose the target shared memory address, and then evict their mapped cache lines to prepare for the second step *Meltdown*, *Meltdown* will illegally access kernel's secret data and make the data stored in the cache, and the third step *Reload* will leak the cached data. The first attack step and the third attack step of *Evict+Reload with Meltdown* are nearly the same as the

TABLE 7. The PAP and *RiskScore* of the *Evict+Time* attack.

Security Mechanisms	P(AS1) / F1	P(AS2) / F2	P(AS3) / F3	PAP / RiskScore
No Security Mechanisms	1.0 / 2.8	1.0 / 3.3	1.0 / 2.8	1.0 / 8.9
No *clflush*	1.0 / 2.8	1.0 / 3.3	1.0 / 2.8	1.0 / 8.9
No shared memory	1.0 / 2.8	1.0 / 3.3	1.0 / 2.8	1.0 / 8.9
SP cache	1.0 / 2.8	0.0 / 0.0	0.0 / 0.0	0.0 / 2.8
CAT	1.0 / 2.8	0.0 / 0.0	0.0 / 0.0	0.0 / 2.8
SA cache (Random)	1.0 / 2.8	0.68 / 2.244	1.0 / 2.8	0.68 / 7.844
No high resolution timer	0.5 / 1.4	1.0 / 3.3	0.5 / 1.4	0.25 / 6.1

first attack step and the third attack step of *Evict+Reload with Spectre*. Therefore the PAS of the first step is 1.0, and the PAS of the third attack step is 0.5.

In the second step, there are four operations, the first operation *Meltdown* represents the beginning of the *Meltdown* step, and $P(Meltdown) = 1.0$. The second operation *Re-ordering* will determine if the following instructions can be executed in advance, if the processor supports the Out-of-Order feature ($P(Req2) = 1.0$), the attacker is able to carefully design the code to ensure that there is an exploitable out-of-order execution ($P(Re\text{-}ordering|Req2) = 1.0$), then $P(Re\text{-}ordering) = P(Re\text{-}ordering|Req2) * P(Req2) = 1.0$. After the *Re-ordering* operation, the processor starts executing out of order. During the Out-of-Order execution, the attacker in user mode can access the kernel data illegally and store them into the cache, the operation *Access Victim's Data* may take a few tries, but eventually, it will succeed, therefore the PAO of the operation *Access Victim's Data* is 1.0. Then the permission checking mechanism works, the data has been read will be discarded (Squashing), at the same time, the CPU will raise an exception, $P(Squashing) = 1.0$. As a result, the PAS of the second attack step is 1.0, and $RiskScore = F1 + F2 + F3 = 11.65$.

Evict+Reload with Meltdown attack does not need *clflush* instruction, therefore *No clflush* does not affect the PAP of this attack. But shared memory is necessary for *Evict+Reload with Meltdown* attack, without shared memory, the *dp2* (P(Req2)) in the third step will be 0.0, and the *Re-access* operation will also be 0.0, which is

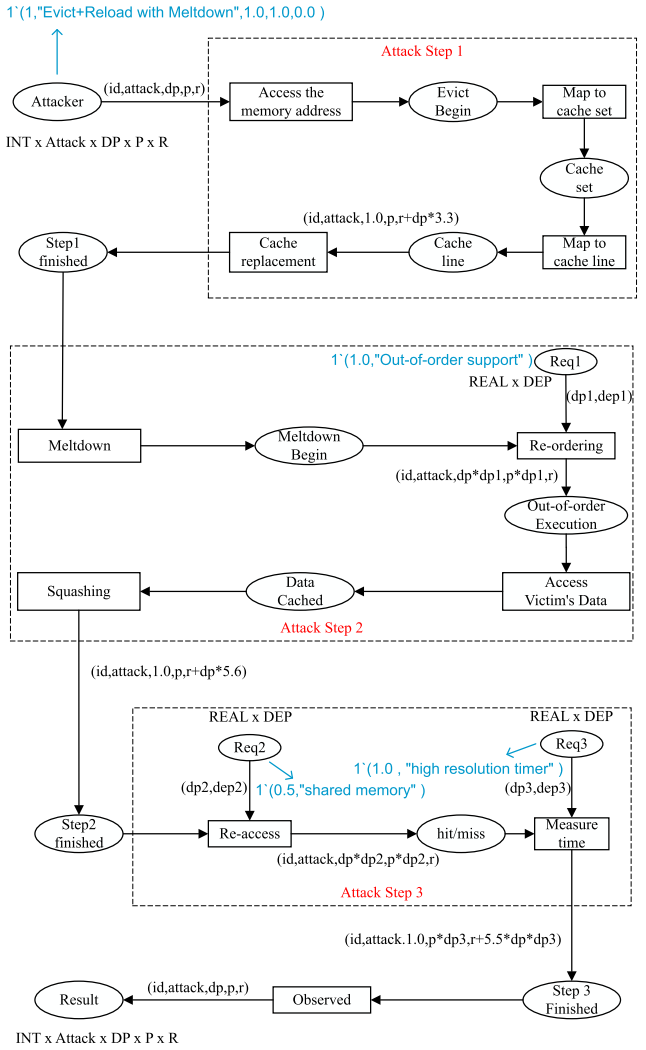


FIGURE 13. Colored petri net model of *Evict+Reload with meltdown* attack

shown in the equation $P(Re\text{-}access) = P(Re\text{-}access|Req2) * P(Req2) = 1.0 * 0.0 = 0.0$.

Same as other *Evict* related attack, *SP cache* and *CAT* based technologies will make the first attack step fail, then the PAS of this step will decrease to 0.0. And SA cache with the random replacement policy will make the PAS of *Evict+Reload with Meltdown* reduce to 0.68.

Due to the multiple methods to build a new timer that meets the accuracy requirements, *No high resolution timer* will not completely defend against the cache side channel attacks, but it can increase the difficulty of the attack. If there is no high resolution timer, *P(Req3)* will be 0.5. Then the PAS of the third attack step will be $P(Re\text{-}access) * P(Measure\ time) = 0.5 * P(Measure\ time|Req3) * P(Req3) = 0.5 * 1.0 * 0.5 = 0.25$.

V. RESULT

We have calculated the PAP and *RiskScore* of six typical cache attacks. To carefully compare the probability and the score, we list them into the Table 9. We mainly focus

TABLE 8. The PAP and *RiskScore* of the *Evict+Reload with meltdown* attack.

Security Mechanisms	P(AS1) / F1	P(AS2) / F2	P(AS3) / F3	PAP / RiskScore
No Security Mechanisms	1.0 / 3.3	1.0 / 5.6	0.5 / 2.75	0.5 / 11.65
No *clflush*	1.0 / 3.3	1.0 / 5.6	0.5 / 2.75	0.5 / 11.65
No shared memory	1.0 / 3.3	1.0 / 5.6	0.0 / 0.0	0.0 / 8.9
SP cache	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0
CAT	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0	0.0 / 0.0
SA cache (Random)	0.68 / 2.244	1.0 / 5.6	0.5 / 2.75	0.34 / 10.594
No high resolution timer	1.0 / 3.3	1.0 / 5.6	0.25 / 1.375	0.25 / 10.275

on the *RiskScore*, which represents the threat level of the cache attacks in the environment with different mechanisms. Besides, PAP is used in the table to indicate whether the security mechanisms are able to effectively reduce the probability of successful attack. In this part, we will analyze the table and answer the two questions left in the section I.

A. THE ANSWER TO THE FIRST QUESTIONS

For the first question that whether there are certain security mechanisms can make the attacker difficult to perform a successful attack?

As Table 9 shown, unfortunately, it is difficult to defend all of the cache attacks by equipping with only one security mechanism, but some combined security mechanisms can defend against all of the cache attacks mentioned in our paper. For example, if we disable the *clflush* instruction, *Flush+Reload* will be defended, and when combined with SP cache or CAT based security mechanisms, other cache attacks will also fail. SP cache and CAT based technologies are the common isolation mechanisms to defend against cache attacks, they are effective and they can successfully defend against most of the typical cache attacks. SP cache is a new cache architecture, and it is difficult to apply to existing computer systems. In contrast, the CAT technology is supported by the Intel processor, and it is lower cost. Therefore, CAT based security mechanisms are worth a try to defend against the side channel attacks.

There are also other security mechanisms can mitigate these cache side channel vulnerabilities. For example, providing SA cache with the random replacement policy or the coarse timer only cannot completely defend against the cache side channel attacks, but they can also make the PAP of the cache attacks decrease. SA cache with the random replacement policy is an effective randomization based method, but the random replacement policy will also degrade computer performance, therefore, ARM only applies the random replacement mechanism to LLC [16], [30]. And beyond that, there are also cache set randomization technologies to defend against the cache side channel attacks, these approaches will make the memory address randomly mapped to the cache set, so that the *Evict* strategy becomes unpractical [31], [32]. These security mitigation techniques are not perfect, but they are sufficient to defend against the real-world cache attacks.

B. THE ANSWER TO THE SECOND QUESTIONS

As for the second question, can we know the degree of threat of the different cache side channel attacks on a system with a specified configuration or security mechanism?

Table 9 shows that when there is no any mechanism, in traditional cache side channel attacks, we need to pay more attention to the *Flush+Reload* attack and the *Prime+Probe* attack, that is because their *RiskScores* are higher. *Flush+Reload* is a high resolution attack, therefore the CVSS score of this attack is high, but it needs *clflush* and *shared memory*, which make the PAP of *Flush+Reload* lower. On the contrary, *Prime+Probe* needs fewer requirements, and it also has sufficient precision, therefore, these two attacks have similar *RiskScore*. And it is worth mentioning that when combined with hardware vulnerabilities, the *RiskScore* of the new *Evict+Reload* attacks (*Evict+Reload with Spectre* or *Evict+Reload with Meltdown*) becomes higher than *Flush+Reload* and *Prime+Probe*.

In the Table 9, we can also easily find which cache attack needs priority attention under the specified security mechanisms. For example, if the computer environment has equipped with SP cache or CAT technologies, most of the cache attacks will be defended. That is because *clflush* instruction is not available in most situations, and *Evict* strategy is widely used in cache attacks to replace the *clflush*. The *Evict* strategy exploits external inference to evict the victim's cache lines, and SP cache and CAT isolate the victim from the attacker and make the victim's cache lines unevictable, which make *Evict* difficult to perform. However, due to the performance considerations, they are used only when the victim is executing important operations. Therefore, when equipped with SP cache or CAT technologies, we need only consider how to protect the data from *Flush+Reload*.

C. LESSONS

In summary, the Table 9 also gives us two important lessons:

1. Cache side channel attacks need some necessary requirements such as shared memory or shared cache lines,

TABLE 9. The PAP and *RiskScore* of the cache side channel attacks.

Defenses / Attacks	Original	Security mechanisms					
	No Security Mechanisms	No *clflush*	No shared memory	SP cache	CAT	SA cache (random)	No high resolution timer
Flush + Reload	0.5/9.95	0.0/0.0	0.0/8.3	0.5/9.95	0.5/9.95	0.5/9.95	0.25/9.125
Evict + Reload	0.5/7.75	0.5/7.75	0.0/6.1	0.0/0.0	0.0/0.0	0.34/6.694	0.25/6.925
Prime + Probe	1.0/9.4	1.0/9.4	1.0/9.4	0.0/0.0	0.0/0.0	0.0039/6.113	0.5/7.75
Evict + Time	1.0/8.9	1.0/8.9	1.0/8.9	0.0/2.8	0.0/2.8	0.68/7.844	0.25/6.1
Evict + Reload with Spectre	0.5/12.35	0.5/12.35	0.0/9.6	0.0/0.0	0.0/0.0	0.34/11.294	0.25/10.975
Evict + Reload with Meltdown	0.5/11.65	0.5/11.65	0.0/8.9	0.0/0.0	0.0/0.0	0.34/10.594	0.25/10.275

there are some security mechanisms that can make these requirements hard to satisfy. However, we should be careful that whether there are other new methods to make the computer environment satisfy these requirements again.

2. Hardware vulnerabilities are getting more and more attention, and if combined with some hardware vulnerability, the traditional cache side channel attacks will have stronger attack power, which may help cache attacks become more threatening real-world attacks. We should take measures to prepare for this in advance.

VI. CONCLUSION

In this paper, we proposed a new colored Petri net method to evaluate the threat level of side channel attacks under certain computer configurations or security mechanisms. There are several advantages to this approach:

1. We consider the requirements and the security weight of the attack steps, which are ignored by other qualitative approaches, so that the score calculated by our method is more reasonable.
2. We adopt the colored Petri net as the modeling tool, which can conveniently build the dependency model of cache attacks. Besides, our method divides the cache side channel attacks into three steps, which also helps us easily model and score each attack step, and the model is easily-extensible. We also adopt the CVSS scoring system to help us score the attack steps more objectively and effectively.
3. Our method is able to show how dangerous are the different cache attacks in the environment with the same security mechanisms. And it also shows the threat level of the same side channel attack under different security mechanisms. These details can help us defend against the cache attacks more effectively at a lower cost.

In the future, we plan to extend the CVSS scoring system to make it more suitable for cache attacks. Cache side channel attacks need many important requirements, which makes it difficult to apply to large-scale attacks in the real world, thus the property *ease of exploiting* plays a more important role in cache side channel attacks than other common types of attacks. However, CVSS is a common scoring system, and some of the metrics are too coarse to reflect the ease of exploiting cache side vulnerabilities. Therefore, improving the CVSS scoring system for cache attacks is also a meaningful work to cope with the increasing risk of cache side channel attacks.

REFERENCES

[1] F. Zhang, Z.-Y. Liang, B.-L. Yang, X.-J. Zhao, S.-Z. Guo, and K. Ren, "Survey of design and security evaluation of authenticated encryption algorithms in the CAESAR competition," *Frontiers Inf. Technol. Electron. Eng.*, vol. 19, no. 12, pp. 1475–1499, 2018.

[2] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. 24th USENIX Secur. Symp. (USENIX Secur.)*, 2015, pp. 897–912.

[3] T. Zhang, F. Liu, S. Chen, and R. B. Lee, "Side channel vulnerability metrics: The promise and the pitfalls," in *Proc. 2nd Int. Workshop Hardw. Architectural Support Secur. Privacy*, 2013, Art. no. 2.

[4] P. Zhou, T. Wang, X. Lou, X. Zhao, F. Zhang, and S. Guo, "Efficient flush-reload cache attack on scalar multiplication based signature algorithm," *Sci. China Inf. Sci.*, vol. 61, no. 3, 2018, Art. no. 039102.

[5] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, and D. Genkin, "Meltdown: Reading kernel memory from user space," in *Proc. 27th USENIX Secur. Symp. (USENIX Secur.)*, 2018, pp. 973–990.

[6] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," 2018, *arXiv:1801.01203*. [Online]. Available: https://arxiv.org/abs/1801.01203

[7] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," Tech. Rep., 2019.

[8] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *Proc. 25th USENIX Secur. Symp. (USENIX Secur.)*, 2016, pp. 565–581.

[9] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *Proc. 39th Annu. Int. Symp. Comput. Architecture (ISCA)*, Jun. 2012, pp. 106–117.

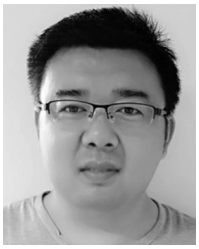
[10] T. Zhang, Y. Zhang, and R. B. Lee, "Analyzing cache side channels using deep neural networks," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 174–186.

[11] T. Zhang and R. B. Lee, "Secure cache modeling for measuring side-channel leakage," Princeton Univ., Princeton, NJ, USA, Tech. Rep., 2014. [Online]. Available: http://palms.ee.princeton.edu/node/428

[12] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2017, pp. 341–353.
[13] S. Deng, W. Xiong, and J. Szefer, "Cache timing side-channel vulnerability checking with computation tree logic," in *Proc. 7th Int. Workshop Hardw. Architectural Support Secur. Privacy*, 2018, Art. no. 2.
[14] Y. Yarom and K. Falkner, "Flush+ Reload: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Secur. Symp. (USENIX Secur.)*, 2014, pp. 719–732.
[15] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 605–622.
[16] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *Proc. 25th USENIX Secur. Symp. (USENIX Secur.)*, 2016, pp. 549–564.
[17] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proc. Cryptographers' Track RSA Conf.* Berlin, Germany: Springer, 2006, pp. 1–20.
[18] C. Percival, "Cache missing for fun and profit," in *Proc. BSDCan*, Ottawa, ON, Canada, 2005. [Online]. Available: http://www.daemonology.net/hyperthreading-considered-harmful/
[19] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 406–418.
[20] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel software guard extensions (intel SGX) support for dynamic memory management inside an enclave," in *Proc. Hardw. Architectural Support Secur. Privacy*, 2016, Art. no. 10.
[21] G. Irazoqui and X. Guo, "Cache side channel attack: Exploitability and countermeasures," *Black Hat Asia*, vol. 2017, no. 3, pp. 1–72, 2017.
[22] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.* Cham, Switzerland: Springer, 2017, pp. 247–267.
[23] P. Vasilikos, H. R. Nielson, F. Nielson, and B. Köpf, "Timing leaks and coarse-grained clocks," in *Proc. IEEE 32nd Comput. Secur. Found. Symp. (CSF)*, Jun. 2019, pp. 1–16.
[24] K. Scarfone and P. Mell, "An analysis of CVSS version 2 vulnerability scoring," in *Proc. 3rd Int. Symp. Empirical Softw. Eng. Meas.*, 2009, pp. 516–525.
[25] *FIRST—Forum of Incident Response and Security Teams CVSS V3.0 Specification Document*. Accessed: Sep. 12, 2019. [Online]. Available: https://www.first.org/cvss/v3.0/specification-document
[26] *FIRST—Forum of Incident Response and Security Teams Common Vulnerability Scoring System Version 3.0 Calculator*. Accessed: Sep. 12, 2019. [Online]. Available: https://www.first.org/cvss/calculator/3.0
[27] W. Tao, Z. Yuqing, Z. Bo, L. Jing, Y. Yunxiang, and G. Jing, "A novel improved system based on CVSS," in *Proc. Int. Conf. Commun., Signal Process., Syst.* Singapore: Springer, 2018, pp. 1015–1021.
[28] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read arbitrary memory over network," in *Proc. Eur. Symp. Res. Comput. Secur.* Cham, Switzerland: Springer, 2019, pp. 279–299.
[29] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen, "CPN tools for editing, simulating, and analysing coloured Petri nets," in *Proc. Int. Conf. Appl. Theory Petri Nets*. Berlin, Germany: Springer, 2003, pp. 450–462.
[30] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite," in *Proc. 42nd Annu. Southeast Regional Conf.*, 2004, pp. 267–272.
[31] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, 2019, pp. 675–692.
[32] W. Song and P. Liu, "Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the LLC," in *Proc. 22nd Int. Symp. Res. Attacks, Intrusions Defenses (RAID)*, 2019, pp. 427–442.

LIMIN WANG was born in Shaoxing, Zhejiang, China, in 1994. He received the B.S. degree from Hangzhou Dianzi University, Hangzhou, China. He is currently pursuing the M.S. degree with the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include computer architecture, cache side channel attack, and formal methods (model checking).

ZIYUAN ZHU was born in Pingdingshan, Henan, China. He received the Ph.D. degree from Tongji University, in 2010. He is currently a Senior Engineer with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. He is currently a Teacher with the School of Cyber Security, University of Chinese Academy of Science, Beijing. His research interests include computer architecture and cyber security.



ZHANPENG WANG received the B.S. degree from the Northeastern University of China, in 2012. He is currently pursuing the Ph.D. degree with the Institute of Information Engineering, Chinese Academy of Sciences. Then, he was with the State Key Laboratory of Robotics, Shenyang Institute of Automation, Chinese Academy of Sciences. His main research interests include computer architecture, side channel analysis, and data protection.

DAN MENG received the B.S., M.S., and Ph.D. degrees from the Harbin Institute of Technology, Harbin, Heilongjiang, China. He was the Director of Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. He is currently the Dean of the School of Cyber Security, University of Chinese Academy of Science, Beijing. His research interests include high-performance computer architecture and cyber security.